Distributed Systems

Group Membership and View
Synchronous Communication

Prof R. Guerraoui
Distributed Programming Laboratory
Group Membership

Who is there?
Group Membership

• In many distributed applications, processes need to know which processes are *participating* in the computation and which are not

• Failure detectors provide such information; however, that information is *not coordinated* (see next slide) even if the failure detector is perfect
Perfect Failure Detector

crash

suspect(p2)
suspect(p2, p3)

p1

p2

p3

p4

suspect(p3) suspect(p2, p3)
suspect(p2, p3)
Group Membership

V1 = (p1, p4)
Group Membership

• To illustrate the concept, we focus here on a group membership abstraction to coordinate the information about **crashes**

• In general, a group membership abstraction can also typically be used to coordinate the processes **joining** and **leaving** explicitly the set of processes (i.e., without crashes)
Group Membership

- **Like** with a failure detector, the processes are informed about failures; we say that the processes *install views*

- **Like** with a perfect failure detector, the processes have accurate knowledge about failures

- **Unlike** with a perfect failure detector, the information about failures are *coordinated*. the processes install the same sequence of views
Group Membership

**Memb1. Local Monotonicity:** If a process installs view (j,M) after installing (k,N), then j > k and M < N

**Memb2. Agreement:** No two processes install views (j,M) and (j,M’) such that M ≠ M’

**Memb3. Completeness:** If a process p crashes, then there is an integer j such that every correct process eventually installs view (j,M) such that p ∉ M

**Memb4. Accuracy:** If some process installs a view (i,M) and p ∉ M, then p has crashed
Group Membership

Events

Indication: <membView, V>

Properties:

- Memb1, Memb2, Memb3, Memb4
Algorithm (gmp)

- **Implements**: groupMembership (gmp).
- **Uses**:
  - PerfectFailureDetector (P).
  - UniformConsensus(Ucons).

- **upon event < Init > do**
  - view := (0,S);
  - correct := S;
  - wait := true;
Algorithm (gmp – cont’d)

upon event < crash, pi > do
  correct := correct \ {pi};

• upon event (correct < view.memb) and (wait = false) do
  • wait := true;
  • trigger<ucPropose,(view.id+1,correct) >;
Algorithm (gmp – cont’d)

- **upon event** < ucDecided, (id, memb)> **do**
  - view := (id, memb);
  - wait := false;
  - **trigger** < membView, view>;}
Algorithm (gmp)

\[
\text{UCons}((p1,p2,p4);(p1,p2,p4))
\]

\[
\text{UCons}((p1,p4);(p1,p4))
\]

\[
\text{UCons}((p1,p3,p4);(p1,p2,p4))
\]

\[
\text{UCons}((p1,p4);(p1,p4))
\]
Group Membership and Broadcast

membView(p1,p3)

p1

p2

membView(p1,p3)

p3

m

crash

m
View Synchrony

- **View synchronous broadcast** is an abstraction that results from the combination of group membership and reliable broadcast.

- **View synchronous broadcast** ensures that the delivery of messages is coordinated with the installation of views.
View Synchrony

Besides the properties of *group membership* (*Memb1*-*Memb4*) and *reliable broadcast* (*RB1*-*RB4*), the following property is ensured:

**VS:** A message is *vsDelivered* in the view where it is *vsBroadcast*
View Synchrony

Events

Request:

<vsBroadcast, m>

Indication:

• <vsDeliver, src, m>
• <vsView, V>
View Synchrony

If the application keeps `vsBroadcasting` messages, the view synchrony abstraction might never be able to `vsInstall` a new view; the abstraction would be impossible to implement.

We introduce a specific event for the abstraction to `block` the application from `vsBroadcasting` messages; this only happens when a process crashes.
View Synchrony

Events

Request:

<vsBroadcast, m>; <vsBlock, ok>

Indication:

<vsDeliver, src, m>; <vsView, V>; <vsBlock>
Algorithm (vsc)

**Implements:** ViewSynchrony (vs).

**Uses:**
- GroupMembership (gmp).
- TerminatingReliableBroadcast (trb).
- BestEffortBroadcast (beb).
upon event < Init > do

view := (0,S); nextView := ⊥;
pending := delivered := trbDone := ∅;
flushing := blocked := false;
Algorithm (vsc – cont’d)

upon event <vsBroadcast,m> and (blocked = false) do

    delivered := delivered ∪ { m };

    trigger <vs Deliver, self, m>;

    trigger <bebBroadcast, [Data,view.id,m]>;
Algorithm (vsc – cont’d)

upon event<bebDeliver,src,[Data,vid,m]) do

If(view.id = vid) and (m ∉ delivered) and (blocked = false) then

 delivered := delivered ∪ { m }

 trigger <vs Deliver, src, m >;
Algorithm (vsc – cont’d)

upon event < membView, V > do
  addtoTail (pending, V);

upon (pending ≠ ∅) and (flushing = false) do
  nextView := removeFromhead (pending);
  flushing := true;
  trigger < vsBlock >;
Algorithm (vsc – cont’d)

Upon <vsBlockOk> do
blocked := true;
trbDone := Ø;
trigger <trbBroadcast, self, (view.id,delivered)>;
Algorithm (vsc – cont’d)

Upon $<\text{trbDeliver}, p, (\text{vid, del})>$ do

$\text{trbDone} := \text{trbDone} \cup \{p\}$;

forall $m \in \text{del}$ and $m \notin \text{delivered}$ do

$\text{delivered} := \text{delivered} \cup \{m\}$;

$\text{trigger} <\text{vsDeliver, src, m}>;$
Upon (trbDone = view.memb) and (blocked = true) do

view := nextView;
flushing := blocked := false;
delivered := Ø;
trigger <vsView, view> ;
Consensus-Based View Synchrony

Instead of launching parallel instances of TRBs, plus a group membership, we use one consensus instance and parallel broadcasts for every view change.

Roughly, the processes exchange the messages they have delivered when they detect a failure, and use consensus to agree on the membership and the message set.
Algorithm 2 (vsc)

- **Implements:** ViewSynchrony (vs).

- **Uses:**
  - UniformConsensus (uc).
  - BestEffortBroadcast(beb).
  - PerfectFailureDetector(P).
Algorithm 2 (vsc – cont’d)

\[\text{upon event } < \text{ Init } > \text{ do}\]

- view := (0,S);
- correct := S;
- flushing := blocked := false;
- delivered := dset := \emptyset;
Algorithm 2 (vsc – cont’d)

 upon event <vsBroadcast,m) and (blocked = false) do

delivered := delivered \cup \{ m \}

 trigger <vsDeliver, self,m>;

 trigger <bebBroadcast,[Data,view.id,m] >;
Algorithm 2 (vsc – cont’d)

upon event <bebDeliver, src, [Data, vid, m]) do

  if (view.id = vid) and (m \notin delivered) and (blocked = false) then
    delivered := delivered \cup \{ m \};

  trigger <vsDeliver, src, m >;
upon event < crash, p > do
  correct := correct \ p ;
  if flushing = false then
    flushing := true;
    trigger <vsBlock> ;
Algorithm 2 (vsc – cont’d)

Upon <vsBlockOk> do

bounced := true;

ttrigger <bebBroadcast, [DSET,view.id,delivered] >;
Algorithm 2 (vsc – cont’d)

Upon <bebDeliver, src, [DSET,vid,del] > do

dset := dset ∪ (src,del);

if forall p ∈ correct, (p,mset) ∈ dset then
  trigger <ucPropose, view.id+1, correct, dset >;
Algorithm 2 (vsc – cont’d)

Upon <ucDecided, id, memb, vsdset> do
  forall (p,mset) ∈ vsdset: p ∈ memb do
  forall (src,m) ∈ mset: m ∉ delivered do
    delivered := delivered ∪ {m}
    trigger <vsDeliver, src, m>;
  view := (id, memb); flushing := blocked := false; dset := delivered := ∅;
  trigger <vsView, view>;
Uniform View Synchrony

We now combine the properties of

*group membership (Memb1-Memb4)* – which is already uniform

*uniform reliable broadcast (RB1-RB4)* –

**VS:** A message is *vsDelivered* in the view where it is *vsBroadcast* – which is already uniform
Uniform View Synchrony

Using uniform reliable broadcast instead of best effort broadcast in the previous algorithms does not ensure the uniformity of the message delivery
Uniformity?

\[ \text{vsView}(p1,p3) \]

\[ \text{vsDeliver}(m) \]

\[ \text{crash} \]
Algorithm 3 (uvsc)

upon event < Init > do
  view := (0,S);
  correct := S;
  flushing := blocked := false;
  udelivered := delivered := dset := \emptyset;
  for all m: ack(m) := \emptyset;
Algorithm 3 (uvsc – cont’d)

upon event <vsBroadcast,m) and (blocked = false) do
  delivered := delivered ∪ \{m\};
  trigger <bebBroadcast,[Data,view.id,m] >;
Algorithm 3 (uvsc – cont’d)

\[
\text{upon event} <\text{bebDeliver}, \text{src}, [\text{Data}, \text{vid}, m]) \text{ do}
\]

\[
\text{if} \ (\text{view.id} = \text{vid}) \text{ then}
\]

\[
\text{ack}(m) := \text{ack}(m) \cup \{\text{src}\};
\]

\[
\text{if} \ m \not\in \text{delivered} \text{ then}
\]

\[
\text{delivered} := \text{delivered} \cup \{m\}
\]

\[
\text{trigger} <\text{bebBroadcast}, [\text{Data}, \text{view.id}, m] >;
\]
Algorithm 3 (uvsc – cont’d)

upon event (view ≤ ack(m)) and (m ∉ udelivered) do

udelivered := udelivered ∪ { m }

trigger <vsDeliver, src(m), m >;
Algorithm 3 (uvsc – cont’d)

upon event < crash, p > do
  correct := correct \{ p \};
  if flushing = false then
    flushing := true;
    trigger <vsBlock>;

Algorithm 3 (uvsc – cont’d)

Upon <vsBlockOk> do

  blocked := true;

  trigger <bebBroadcast, [DSET, view.id, delivered] >;

Upon <bebDeliver, src, [DSET, vid, del] > do

dset := dset ∪ (src, del);

if forall p ∈ correct, (p, mset) ∈ dset then trigger <ucPropose, view.id+1, correct, dset >;
Algorithm 3  (uvsc – cont’d)

Upon <ucDecided, id, memb, vsdset> do
  forall (p,mset) ∈ vs-dset: p ∈ memb do
    forall (src,m) ∈ mset: m ∉ udelivered do
      udelivered := udelivered ∪ \{m\}
    trigger <vsDeliver, src, m>;
  view := (id, memb); flushing := blocked := false; dset := delivered := udelivered := ∅;
  trigger <vsView, view>;